为什么要使用 Redux

React 有 props 和 state:

1. props 意味着父级分发下来的属性
2. state 意味着组件内部可以自行管理的状态，并且整个 React 没有数据向上回溯的能力，这就是 react 的单向数据流

这就意味着如果是一个数据状态非常复杂的应用，更多的时候发现 **React 根本无法让两个组件互相交流**，使用对方的数据，react 的通过层级传递数据的这种方法是非常难受的，这个时候，迫切需要一个机制，**把所有的 state 集中到组件顶部，能够灵活的将所有 state 各取所需的分发给所有的组件**，是的，这就是 redux

简介

1. redux 是的诞生是为了给 React 应用提供「可预测化的状态管理」机制。
2. Redux 会将整个应用状态(其实也就是数据)存储到到一个地方，称为 store
3. 这个 store 里面保存一棵状态树(state tree)
4. 组件改变 state 的唯一方法是通过调用 store 的 dispatch 方法，触发一个 action，这个 action 被对应的 reducer 处理，于是 state 完成更新
5. 组件可以派发(dispatch)行为(action)给 store,而不是直接通知其它组件
6. 其它组件可以通过订阅 store 中的状态(state)来刷新自己的视图

使用步骤

1. **创建 reducer**
    - 可以使用单独的一个 reducer,也可以将多个 reducer 合并为一个 reducer，即：combineReducers()
    - action 发出命令后将 state 放入 reucer 加工函数中，返回新的 state,对 state 进行加工处理
2. **创建 action**
    - 用户是接触不到 state 的，只能有 view 触发，所以，这个 action 可以理解为指令，需要发出多少动作就有多少指令
    - action 是一个对象，必须有一个叫 type 的参数，定义 action 类型
3. **创建的 store，使用 createStore 方法**
    - store 可以理解为有多个加工机器的总工厂

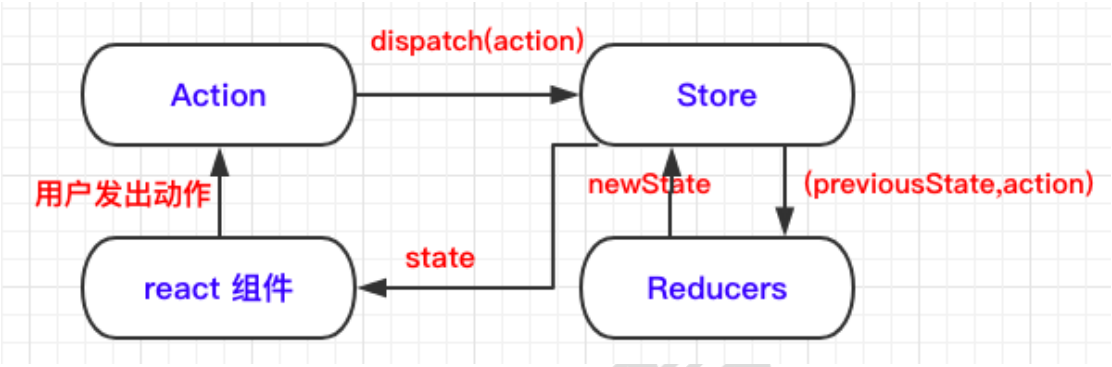
- 提供 subscribe，dispatch，getState 这些方法。

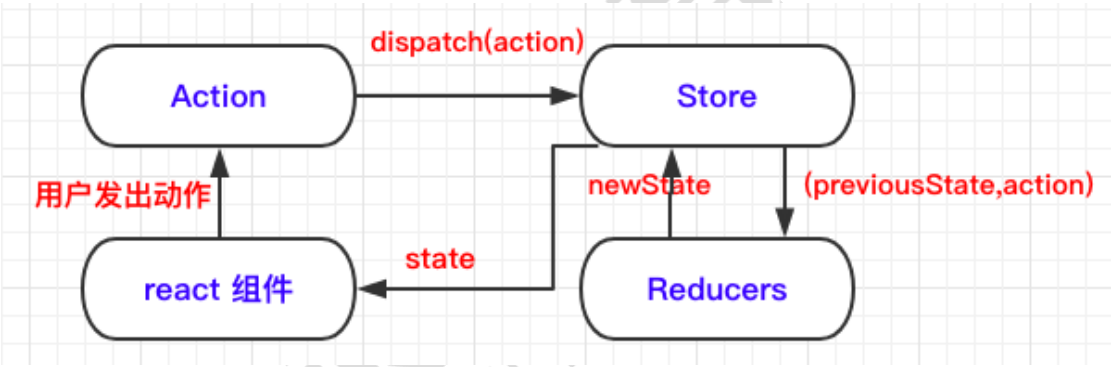
按步骤手把手实战。

上述步骤，对应的序号，我会在相关代码标出

```
npm install redux -S // 安装

import { createStore } from 'redux' // 引入

const reducer = (state = {count: 0}, action) => {----------> (1)
  switch (action.type){
    case 'INCREASE': return {count: state.count + 1};
    case 'DECREASE': return {count: state.count - 1};
    default: return state;
  }
}

const actions = {---------->(2)
  increase: () => ({type: 'INCREASE'}),
  decrease: () => ({type: 'DECREASE'})
}

const store = createStore(reducer);---------->(3)

store.subscribe(() =>
  console.log(store.getState())
);

store.dispatch(actions.increase()) // {count: 1}
store.dispatch(actions.increase()) // {count: 2}
store.dispatch(actions.increase()) // {count: 3}
```

自己画了一张非常简陋的流程图，方便理解 redux 的工作流程





react-redux

刚开始就说了，如果把 store 直接集成到 React 应用的顶层 props 里面，只要各个子组件能访问到顶层 props 就行了，比如这样：

```
<顶层组件 store={store}>
  <App />
</顶层组件>
```

不就 ok 了吗？这就是 react-redux。Redux 官方提供的 React 绑定库。 具有高效且灵活的特性。

React Redux 将组件区分为 容器组件 和 UI 组件

1. 前者会处理逻辑
2. 后者只负责显示和交互，内部不处理逻辑，状态完全由外部掌控

两个核心

- Provider

  看我上边那个代码的**顶层组件** 4 个字。对，你没有猜错。这个顶级组件就是 Provider,一般我们都将顶层组件包裹在 Provider 组件之中，这样的话，所有组件就都可以在 react-redux 的控制之下了,**但是 store 必须作为参数放到 Provider 组件中去**

  ```
  <Provider store = {store}>
      <App />
  <Provider>
  ```

  这个组件的目的是让所有组件都能够访问到 Redux 中的数据。

- connect

  这个才是 react-redux 中比较难的部分，我们详细解释一下

  首先，先记住下边的这行代码：

  ```
  connect(mapStateToProps, mapDispatchToProps)(MyComponent)
  ```

  mapStateToProps

  这个单词翻译过来就是**把 state 映射到 props 中去** ,其实也就是**把 Redux 中的数据映射到 React 中的 props 中去。**

  举个栗子：

```
  const mapStateToProps = (state) => {
    return {
      // prop : state.xxx  | 意思是将 state 中的某个数据映射到 props 中
       foo: state.bar
    }
  }
```

然后渲染的时候就可以使用 this.props.foo

```
class Foo extends Component {
    constructor(props){
```

```
        super(props);
    }
    render(){
        return(
                // 这样子渲染的其实就是 state.bar 的数据了
            <div>this.props.foo</div>
        )
    }
}
Foo = connect()(Foo);
export default Foo;
```

然后这样就可以完成渲染了

mapDispatchToProps

这个单词翻译过来就是就是**把各种 dispatch 也变成了 props 让你可以直接使用**

```
const mapDispatchToProps = (dispatch) => { // 默认传递参数就是
dispatch
  return {
    onClick: () => {
      dispatch({
        type: 'increatment'
      });
    }
  };
}

class Foo extends Component {
    constructor(props){
        super(props);
    }
    render(){
        return(

             <button onClick = {this.props.onClick}>点击
increase</button>
        )
    }
}
Foo = connect()(Foo);
export default Foo;
```

组件也就改成了上边这样，可以直接通过 this.props.onClick，来调用 dispatch,这样子就不需要在代码中来进行 store.dispatch 了

react-redux 的基本介绍就到这里了

redux-saga

如果按照原始的 redux 工作流程，当组件中产生一个 action 后会直接触发 reducer 修改 state，reducer 又是一个纯函数，也就是不能再 reducer 中进行异步操作；

而往往实际中，组件中发生的 action 后，在进入 reducer 之前需要完成一个异步任务,比如发送 ajax 请求后拿到数据后，再进入 reducer,显然原生的 redux 是不支持这种操作的

这个时候急需一个中间件来处理这种业务场景，目前最优雅的处理方式自然就是 redux-saga

核心讲解

1、Saga 辅助函数

redux-saga 提供了一些辅助函数，用来在一些特定的 action 被发起到 Store 时派生任务，下面我先来讲解两个辅助函数：takeEvery 和 takeLatest

- takeEvery

takeEvery 就像一个流水线的洗碗工，过来一个脏盘子就直接执行后面的洗碗函数，一旦你请了这个洗碗工他会一直执行这个工作，绝对不会停止接盘子的监听过程和触发洗盘子函数

例如：每次点击 按钮去 Fetch 获取数据时时，我们发起一个 FETCH_REQUESTED 的 action。 我们想通过启动一个任务从服务器获取一些数据，来处理这个 action，类似于

```
window.addEventLister('xxx',fn)
```

当 dispatch xxx 的时候，就会执行 fn 方法，

首先我们创建一个将执行异步 action 的任务(也就是上边的 fn)：

```
// put：你就认为 put 就等于 dispatch 就可以了；
```

```
// call：可以理解为实行一个异步函数,是阻塞型的，只有运行完后面的函数，才会继续往下；
// 在这里可以片面的理解为 async 中的 await！但写法直观多了！
import { call, put } from 'redux-saga/effects'

export function* fetchData(action) {
   try {
      const apiAjax = (params) => fetch(url, params);
      const data = yield call(apiAjax);
      yield put({type: "FETCH_SUCCEEDED", data});
   } catch (error) {
      yield put({type: "FETCH_FAILED", error});
   }
}
```

然后在每次 FETCH_REQUESTED action 被发起时启动上面的任务,也就**相当于每次触发一个名字为 FETCH_REQUESTED 的 action 就会执行上边的任务**,代码如下

```
import { takeEvery } from 'redux-saga'

function* watchFetchData() {

  yield* takeEvery("FETCH_REQUESTED", fetchData)
}
```

注意：上面的 takeEvery 函数可以使用下面的写法替换

```
function* watchFetchData() {

   while(true){
     yield take('FETCH_REQUESTED');
     yield fork(fetchData);
   }
}
```

- takeLatest

在上面的例子中，takeEvery **允许多个 fetchData 实例同时启动**，在某个特定时刻，我们可以启动一个新的 fetchData 任务， 尽管之前还有一个或多个 fetchData 尚未结束

如果我们**只想得到最新那个请求的响应**（例如，始终显示最新版本的数据），我们可以使用 takeLatest 辅助函数

```
import { takeLatest } from 'redux-saga'

function* watchFetchData() {
  yield* takeLatest('FETCH_REQUESTED', fetchData)
}
```

和 takeEvery 不同，在任何时刻 takeLatest 只允许执行一个 fetchData 任务，并且这个任务是最后被启动的那个，如果之前已经有一个任务在执行，那之前的这个任务会自动被取消

2、Effect Creators

redux-saga 框架提供了很多创建 effect 的函数，下面我们就来简单的介绍下开发中最常用的几种

- take(pattern)
- put(action)
- call(fn, ...args)
- fork(fn, ...args)
- select(selector, ...args)

take(pattern)

take 函数可以理解为监听未来的 action，它创建了一个命令对象，告诉 middleware 等待一个特定的 action， Generator 会暂停，直到一个与 pattern 匹配的 action 被发起，才会继续执行下面的语句，也就是说，take 是一个阻塞的 effect

用法：

```
function* watchFetchData() {
   while(true) {
   // 监听一个 type 为 'FETCH_REQUESTED' 的 action 的执行，直到等到这个 Action 被触发，才会接着执行下面的 yield fork(fetchData) 语句
     yield take('FETCH_REQUESTED');
     yield fork(fetchData);
   }
}
```

put(action)

put 函数是用来发送 action 的 effect，你可以简单的**把它理解成为 redux 框架中的 dispatch 函数**，当 put 一个 action 后，reducer 中就会计算新的 state 并返回，**注意： put 也是阻塞 effect**

用法：

```
export function* toggleItemFlow() {
    let list = []
    // 发送一个 type 为 'UPDATE_DATA' 的 Action，用来更新数据，参数为 `data：list`
    yield put({
      type: actionTypes.UPDATE_DATA,
      data: list
    })
}
```

call(fn, ...args)

call 函数你可以把它简单的理解为就是可以调用其他函数的函数，它命令 middleware 来调用 fn 函数， args 为函数的参数，**注意： fn 函数可以是一个 Generator 函数，也可以是一个返回 Promise 的普通函数**，call 函数也是**阻塞 effect**

用法：

```
export const delay = ms => new Promise(resolve => setTimeout(resolve, ms))

export function* removeItem() {
  try {
    // 这里 call 函数就调用了 delay 函数，delay 函数为一个返回 promise 的函数
    return yield call(delay, 500)
  } catch (err) {
    yield put({type: actionTypes.ERROR})
  }
}
```

fork(fn, ...args)

fork 函数和 call 函数很像，**都是用来调用其他函数的，但是 fork 函数是非阻塞函数**，也就是说，**程序执行完 yield fork(fn， args) 这一行代码后，会立即接着执行下一行代码语句，而不会等待 fn 函数返回结果后**，在执行下面的语句

用法：

```
import { fork } from 'redux-saga/effects'

export default function* rootSaga() {
  // 下面的四个 Generator 函数会一次执行，不会阻塞执行
  yield fork(addItemFlow)
  yield fork(removeItemFlow)
  yield fork(toggleItemFlow)
  yield fork(modifyItem)
}
```

select(selector, ...args)

select 函数是用来指示 middleware 调用提供的选择器获取 Store 上的 state 数据，你也可以简单的把它理解为 **redux 框架中获取 store 上的 state 数据一样的功能** ： store.getState()

用法：

```
export function* toggleItemFlow() {
     // 通过 select effect 来获取 全局 state 上的 `getTodoList` 中的 list
     let tempList = yield select(state => state.getTodoList.list)
}
```